A Crash Course in (bash) Shell Scripting

KEY

Normal text

`Code`

Key points/common mistakes

Joyce Zhu

7/23/2014

Introduction

Learning shell scripting can greatly improve your productivity and the speed of your experiments. Shell scripts can submit large numbers of jobs in parallel and do long and repetitive text processing jobs in seconds. This tutorial aims to quickly teach you write shell scripts which will help you, a scientist and researcher, accomplish tasks more efficiently.

Before we get started, make sure you're comfortable with the following….

What this is NOT a tutorial on/You need to be familiar with:

- Remote desktop/SSH/the SLURM job submission software: See https://rc.fas.harvard.edu/resources/odyssey-quickstart-guide/ .
- Programming in general (program design, conditionals, iteration) and Python: You don't HAVE to know Python to write shell scripts, but calling it from your shell scripts makes doing a variety of things easier. If you are a Python novice or have never programmed before, head over to http://learnpythonthehardway.org/book.
- Basic Unix commands: If you've never sat at a terminal before, go to https://software.rc.fas.harvard.edu/training/intro_unix/latest. Get comfortable with the stuff up to slide 40 and keep this link on hand for reference if anything later is unfamiliar. Furthermore, find a text editor you like. Doesn't matter if you're a fellow vi enthusiast, an emacs cult member, or you just use gedit because you like GUIs.

What's the Difference Between the Command Line and Shell Scripting?

The commands in a shell script are executed in turn as if you'd typed them into the command line yourself. You just need to type the name of the shell script (you may need to put a ./ in front of the name to clarify to the shell that it's in the current directory) into the command line. Obviously, this makes writing a shell script useful for automating repetitive tasks. **The one difference you need to be aware of is that the first line of your script MUST be:**

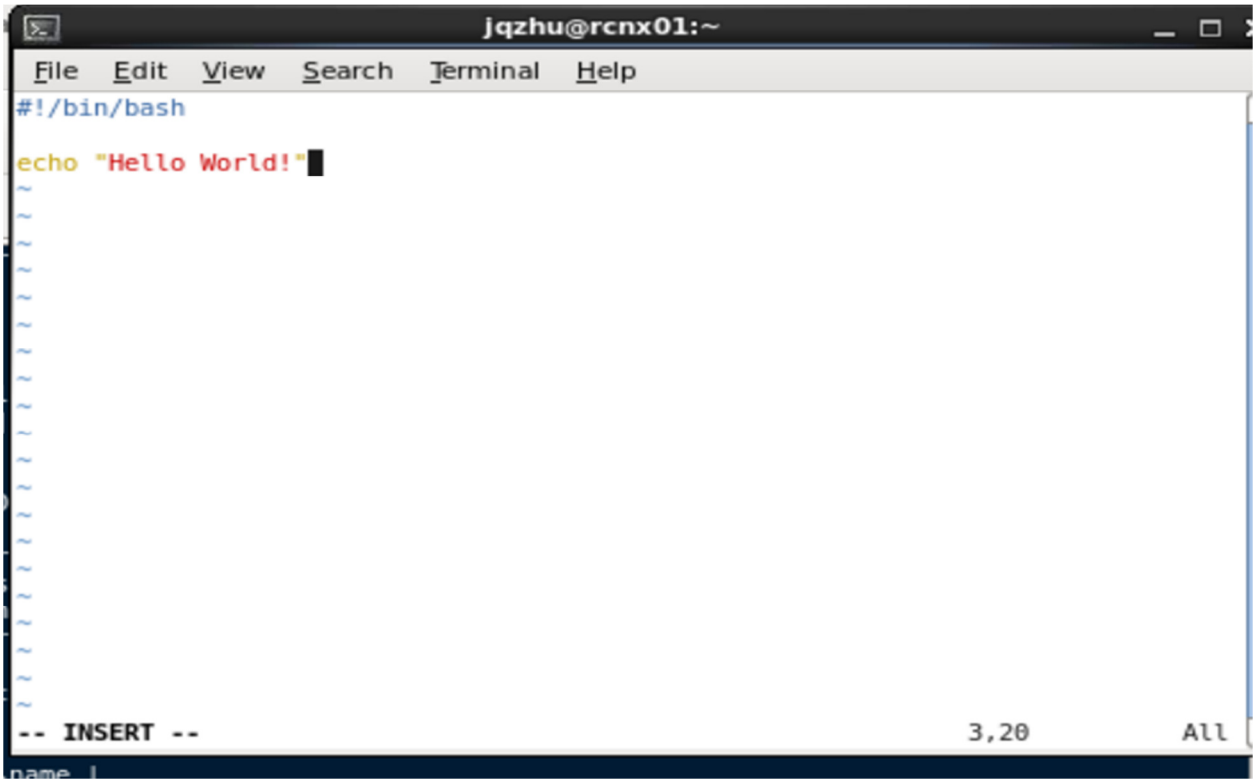
```
#!/bin/bash
```

This directs the bash shell (you shouldn't have to use other shells) to interpret the commands. Also, although this isn't technically necessary, **you should name your shell scripts ending with .sh.** This makes manipulating large numbers of scripts much easier.

The Basics: Printing, Comments, Assigning Variables

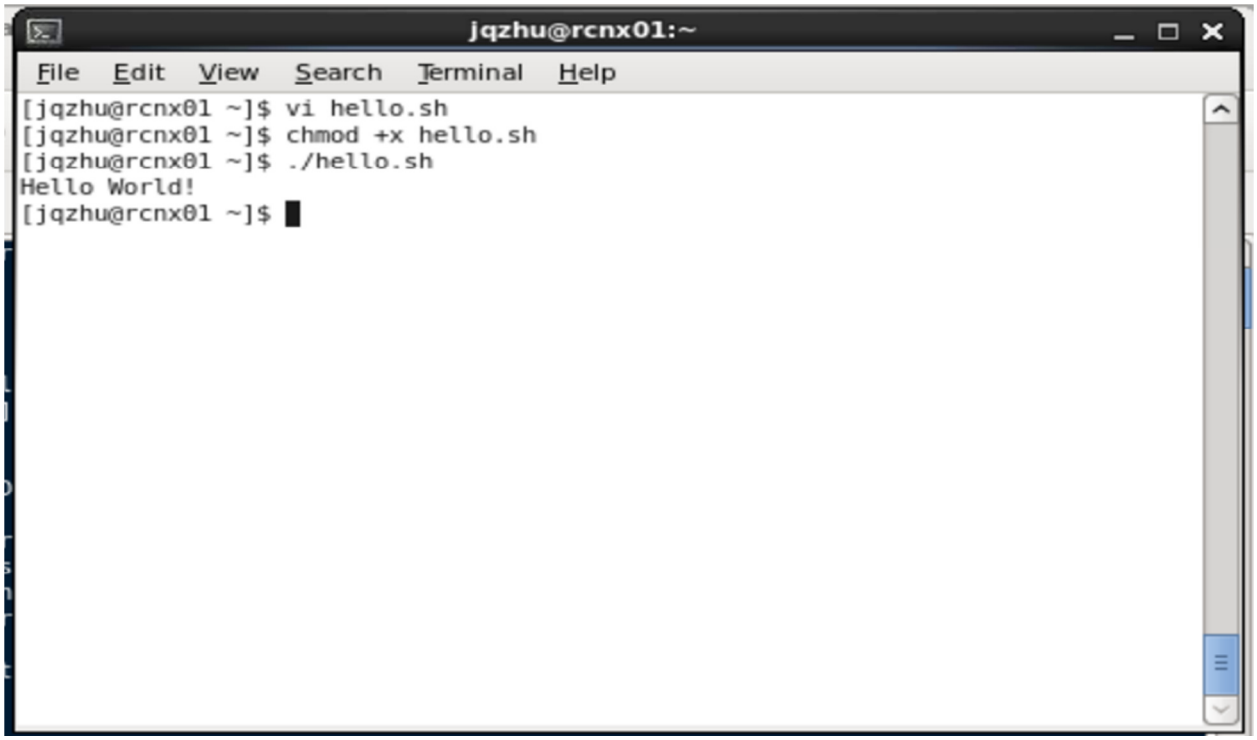
The shell's equivalent of a print statement is the `echo` command. Get ready to write a "hello world" program – open up your text editor and type the following:

```
#!/bin/bash
echo "hello world!"
```



An example in vi, running Linux CentOS 6 on the FAS Odyssey remote desktop.

If this script is named hello.sh, you can then type `./hello.sh` into the terminal to get greeted by "hello world!" (If you get "permission denied", type `chmod +x hello.sh` and try again.)



Your terminal might look something like this when you write and then execute your 1st shell script.

That's your first and last example program. Now that you know how to write up and run a program, I'm going to quickly go over the syntax for various programming constructs.

Any line with a pound sign in front of it is a comment, aka not treated as code by the shell (with the exception of that first line telling bash to interpret your script).

Assigning variables just uses a single = sign without spaces on either side. Eg:

```
total=10
password=iloveprogramming
foo=$(echo $bar)
```
←This $() grouping instructs bash to execute the commands inside the parentheses first, then assign the result to a variable.

When you refer to these variables later, use the $ sign in front of their names. `echo total` just prints out "total", whereas `echo $total` prints out 10 as assigned above.

To assign or update variables involving **simple INTEGER arithmetic**, use the let statement:

`let "total=total+1"` **OR** you can use double parentheses:

```
(( bananas = 3*$apples ))
```

You can also have array/list/enum variables (which have indices):

```
FRUIT=([1]="cherry" [2]="watermelon" [3]="mango")
```

When referring to the value in a **specific index, you need to use curly braces**, not parentheses:

```
echo ${FRUIT[2]}
```

Metacharacters

Unix has a number of metacharacters. The ones which will be useful in shell scripts are:

- >: Output redirection. `program.py > out.txt` will write print statements from your Python program into a text file instead of printing them to the terminal screen.
- <: Input redirection. Does the opposite. Good for sending program pre-prepared input.
- |: **Pipe**. One of the most powerful parts of Unix – you can string several utilities together. `who | sort`, for example, outputs a sorted list of users. (The pipe character is Shift + \.)

We'll discuss *, . , [ ], $, and \ in the sed/regex section.

Syntax for Iteration, Conditionals, and Waiting (sleep)

There are a number of ways to write "if" statements and they often differ based on which shell you're using. In fact, the bash beginner's guide has an entire chapter on it: http://www.tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

The syntax I'd recommend is the following **(make sure to leave spaces between the condition and the double brackets)**:

```
if [[ condition ]]; then
    #do stuff
fi
```

The tldp link above contains a reference to the syntax for checking numerous conditions, such as if a file is a directory (`-d filename`), operation one is not equal to operation two (`op1 -ne op2`), and if a file exists (`-e filename`).

There are two main ways to write a "for" loop. Using the seq (sequence) command is often the simplest. The following will print numbers 1 through 5 **(make sure you're using grave accents, not single quotes, when using seq)** – look on the top left of the keyboard:

```
for i in `seq 1 5`; do echo $i; done
```

There is also a "for in" style that will be familiar to Java and Python programmers:

```
for x in $foo/*;
do
   #something
done
```

will do something for every directory in location foo (we'll talk about that asterisk later).

"while" syntax is similar:

```
while [[ condition ]];
do
     #something
done
```

If you want your script to wait for a certain amount of time before continuing, use the sleep command. This is especially useful when you can only continue given a certain condition – for example, you need a program to finish running before you call a post-processing script. **The number you type after "sleep" is the number of seconds the script will pause**. The following example pauses 5 seconds between printing directory names:

```
cd foo; sleep 5; pwd; cd ..; sleep 5; pwd
```

Handling Floating Point Numbers in the Shell

If you're not working with integers you'll quickly find that the shell does a terrible job of handling floating point numbers. 1 + 0.5 still equals 1! There are two ways around this.

- Call another program/language from the command line; I like Python, but you can use Ruby, Perl, etc. The Python syntax for storing 1 + 0.5 in a variable and getting 1.5 is:

  ```
  foo=$(python -c "print 1 + 0.5")
  ```

  This code translates to, "Run the code within the double quotes in Python, then store the result in a shell variable called foo".

  In general, if doing something in Python is easier than doing it in shell, you can use that Python –c syntax to output the result of running the double-quoted code in Python and store it in a shell variable. Try to limit this as much as possible; **calling Python for the ENTIRE program means you might as well have written a Python script!**

- Use "echo" to pipe into the shell utility bc (basic calculator). **Specify the scale variable for how much decimal place accuracy you'd like**. The example below will yield 2.0005. `example=$(echo "scale=5;1 + 1.0005" | bc)`
  bc has a lot of functionality; if you ever need it to do more, here's a complete guide: http://www.basicallytech.com/blog/?/archives/23-command-line-calculations-using-bc.html

Debugging Tips

- **Echo variables** every time they need to change or you just want to check the value.
- To stop the script after a certain point, just type exit after the last line to be executed.
- Check over syntax. Did you miss a bracket? Forget a semicolon before typing "then"?
- Use the red and bold lines as a checklist, and check your code against my code samples.

Armed with the above knowledge, you can already write useful little shell scripts. If you're here to learn about how to efficiently change text in files, though, stick around and learn about sed.

(Advanced) Using the sed Command Line Tool

sed stands for "stream editor"; it is a powerful Unix text processing tool which is great for automating repetitive tasks. You can use it to scan files and perform actions on all lines which match a certain criteria. For example, if you want to insert some spaces at the front of each line in a file, you can just type:

```
sed -i 's/^/  /' foo.txt
```

instead of going into a text editor and doing it yourself. You probably want to learn at least the ropes of using sed because **sed will allow you to generate and run the same script multiple times with different values for certain variables**.

The basic sed syntax you'll need to know is:

```
sed [-i] 's/foo/bar/' example
```

This will replace all instances of the string foo in the file example with the string bar. The **-i option stands for "in-place change";** if you don't use it, sed won't change the file directly. It will instead print to the terminal screen what the file would look like *if* it were changed, but if you open the file itself, you'll notice that it remains unchanged.

If you're typing directly into the command line, put either single or double quotes around your sed script. Eg, `sed -i 's/^/  /' foo.txt` . If you're using sed in a bash script, you don't need the quotes. In either case, **remember to escape nested quotes if you're writing quotes to the (output) file itself.** (Escaping a character entails putting a backslash in front of it. If you've never seen this before, I'll elaborate a bit more in this following section on regex.

(Advanced) Introduction to Regular Expressions

Regex is short for "regular expressions"; basically, using regex entails using certain characters, called metacharacters, to describe the types of strings you want to match. This is an incredibly timesaving tool, but also can get insanely complex. You probably only need to know the following handful of metacharacters:

--^ : match only the beginning of a line

--$: match only the end of a line

--. : match any single character

--* : matches zero or more occurrences of the character it precedes. Remember the asterisk in the section about for loops? If you go back, you'll now realize that the syntax matches any directory due to the use of this asterisk after the / character. So **to search for foo but replace the entire line after finding it, write foo.***

--[]: match any *single* character within the brackets. Eg, [12345] matches anything containing 1, 2, 3, 4, or 5. You can use a hyphen to specify a range; [a-zA-Z] matches any one letter.

--\ : use before any metacharacters you are actually searching for in a file. For example, if you are trying to match an array, make sure to **escape (put a backslash before)** each of the square brackets, or sed will treat the brackets as part of a regular expression!

The * metacharacter is especially important since **you can use it to do simple repeated operations to similar files**. For example, if you have a number of shell scripts in a directory you want to submit to sbatch and you've followed my advice and consistently given your scripts a .sh extension, you can just type `sbatch *.sh` and you're done! (Obviously doing more complex repeated operations, such as file modification and creation, will require shell scripting.)

It takes a while to practice and get used to complex-looking sed and regular expressions, so I've provided a few examples below for your reference and linked in-depth tutorials at the end.

--`sed -i 's/^/  /' foo.txt` : put 2 spaces in front of every line in foo.txt
--`sed -i 's/^ *//' foo.txt` : remove leading spaces in front of every line in foo.txt
--`sed -i s/"Intern1"/"Intern2"/ genericLetter` : change the name of the intern you're sending a letter to
--`sed -i s/"dic\['banana'\]".*/"dic['apple'] = 'fruit'"/ foo.py` : Change a dictionary value in a Python script. Note the backslash to escape the square brackets, which are valid regex metacharacters.

Program Structure for Generating and Submitting Parallel Scripts

You want to strike a balance between minimizing how much you type and having reasonably-sized scripts (aka anything over, say, 200 lines is probably too much). If you have access to my parallel ABAQUS code, you can examine the README and the code to see how it's structured. In any case, I recommend you follow the following layout roughly:

--Have whatever scripts you want to run in parallel ready: a Python file for ABAQUS, a MATLAB script, etc. In any case, you do need a shell script to submit to SBATCH; this script should specify SLURM options (see the Odyssey documentation), load modules, call whatever program you want, and (optionally) collect output and store it in a different file.

--The first script you should actually run from the command line will be a preparation script. Based on user-defined variables, this script should create a subdirectory for each job, put a copy of each of the files defined above into each subdirectory, and use sed to change the values of variables in these files if necessary. (Subdirectories for each job aren't strictly necessary, but do make organizing experiments MUCH easier!) **All creation and changing of files should be done in this first script.**

--The second and last script to run from the command line will be a mass submission script. Use a loop to submit all the shell files to SLURM using the sbatch command. This might require changing directories, but should be nothing complex, since you're just submitting everything with a loop. You already did the complex stuff (creating and modifying everything) beforehand.

That should be all you need to type in; you can go do something else until your jobs complete!

Additional Resources

I've linked to resources where appropriate, but here is some additional help.

http://www.grymoire.com/unix/sed.html -- Excellent tutorial on the intricacies of sed.

http://regex.learncodethehardway.org/book/ -- Regular expressions explained clearly.

http://www.tldp.org/LDP/Bash-Beginners-Guide/html/index.html -- Best in-depth bash guide for beginners online.

http://www.freeos.com/guides/lsst/ch08.html -- Cute practice exercises. The author isn't a native English speaker, so I wouldn't try to use his accompanying tutorial as a Unix guide, but these are good for helping get shell syntax under your fingers.